Efficient Parallel FFTs for Different Computational Models

Efficient Parallel FFTs for Different Computational Models

Nadia Shalaby

TR-17-96

December 1996

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Efficient Parallel FFTs for Different Computational Models*

Nadia Shalaby†

December 25, 1996

Abstract

We select the Fast Fourier Transfrom (FFT) to demonstrate a methodology for deriving the optimal parallel algorithm according to predetermined performance metrics, within a computational model. Following the vector space framework for parallel permutations, we provide a specification language to capture the algorithm, derive the optimal parallel FFT specification, compute the arithmetic, memory, communication and load–balance complexity metrics, apply the analytical performance evaluation to PRAM, LPRAM, BSP and LogP computational models, and compare with actual performance results.

1 Introduction

The FFT algorithm, popularized by Cooley and Tukey [2], performs the discrete Fourier Transform (DFT) in $O(N \log N)$ time, as opposed to the naive $O(N^2)$, and is ubiquitously used in most areas of applied sciences and engineering [8]. As more of these applications migrate to the platform of parallel computing, the exigency of a highly efficient and portable parallel FFT kernel becomes evident. Considerable valuable research was conducted on this topic, such as [5, 6, 7, 9, 15, 16]. However, studies tended to be geared towards specific architectures, or did not fully analyze the system and problem parameters at hand, or gave underspecified algorithms, and overall did not present a general approach for the parallelization of other orthogonal transforms.

The above perspective motivates a number of goals. Our first goal, addressed in § 2, is to develop a methodology and algorithmic specification that can be extended to other parallel orthogonal transforms, such as the discrete cosine transform [11] and the Legendre transform [13] and define a set of complexity metrics suitable for this class of algorithms. The second goal, to derive a parallel FFT algorithm and prove that the choices made are optimal, is covered in § 3 and 4. Third, in § 5, we demonstrate that this computational framework is sufficiently general to render abstraction to computational models such as the PRAM [4], LPRAM [1], BSP [17] and LogP [3]. On the other hand, § 6 illustrates that the specification and analysis are refined enough for immediate implementation, and we present our comparative implementations on the CM–5 as an example.

2 Methodology

Computational Framework: We denote the problem size by $N = 2^r$, the number of processors by $P = 2^p$ and the virtual processor ratio as $V = N/P$. Thus, $p = \log P$,

*Partial support by Office of Naval Research grant N00014-93-1-0192, Air Force Office of Scientific Research grant F49620-93-1-0480 and Los Alamos National Laboratory contract 8283K0014-9U.

†Aiken Computation Lab., Harvard University, Cambridge, MA 02138

$v = \log V$ and $r = \log N = p + v$ represent the number of bits in processor, local memory and data addresses, respectively. All logarithms are base two.

We define a permutation Π on an ordered set A to be a bijective map $\Pi : A \longrightarrow A$ and denote the composition of two permutations as $\Pi_1 \odot \Pi_2$. For simplicity, we assume $N \geq P$ (by setting $u = r$ in [12]), and the machine's physical memory location address is modeled as a vector space Z_r over $\mathbb{F}_2$, where $|Z_r| = 2^r$, and for $P = 2^p$ processors, a vector $z \in Z_r$ is denoted by $z_{r-1} \ldots z_{v+1} z_v \mid z_{v-1} \ldots z_1 z_0$ where the left and right parts represent the *processor* and *memory* addresses respectively (thus arbitrarily selecting *block* allocation). Similarly, the data element index is modeled by a vector space A_r over $\mathbb{F}_2$, where $|A_r| = 2^r$, and a vector $a \in A_r$ is denoted by $a_{r-1} a_{r-2} \ldots a_1 a_0 = (a_i)_{i=r-1}^{0}$. We assume a *uniform mapping* between the set of 2^r indices and the set of all 2^p processors, where each processor is assigned 2^v indices, and assignment is made to local memory addresses 0 to $2^v - 1$.

The set of permutations relevant to the FFT can now be defined within this vector space framework. The bit–reversal permutation, $\Pi_{bitrev}^{m,l}$, is a permutation on A_r such that $\Pi_{bitrev}^{m,l}\left((a_i)_{i=r-1}^{m+1}(a_i)_{i=m}^{l}(a_i)_{i=l-1}^{0}\right) = (a_i)_{i=r-1}^{m+1}(a_i)_{i=l}^{m}(a_i)_{i=l-1}^{0}$.

Let $Q = 2^q$, where $q = 1, 2, \ldots \lfloor r/2 \rfloor$ and let $l, m \in I$ such that $0 \leq l, m \leq r-1$, $m - l \geq q$, $m + q - 1 \leq r - 1$. Then the radix–Q exchange permutation, $\Pi_{xch,Q}^{m,l}$, is a permutation on A_r defined as: $\Pi_{xch,Q}^{m,l}\left((a_i)_{i=r-1}^{m+q}(a_i)_{i=m+q-1}^{m}(a_i)_{i=m-1}^{l+q}(a_i)_{i=l+q-1}^{l}(a_i)_{i=l-1}^{0}\right) = (a_i)_{i=r-1}^{m+q}(a_i)_{i=l+q-1}^{l}(a_i)_{i=m-1}^{l+q}(a_i)_{i=m+q-1}^{m}(a_i)_{i=l-1}^{0}$.

Let $a \in A_r$, and let $0 \leq l, m \leq r-1$ and $Q = 2^q$. Then the radix–Q shuffle permutation, $\Pi_{shf,Q}^{m,l}(a)$, is defined as: $\Pi_{shf,Q}^{m,l}\left((a_i)_{i=r-1}^{m+q}(a_i)_{i=m}^{m-q+1}(a_i)_{i=m-q}^{l}(a_i)_{i=l-1}^{0}\right) = (a_i)_{i=r-1}^{m+q}(a_i)_{i=m-q}^{l}(a_i)_{i=m}^{m-q+1}(a_i)_{i=l-1}^{0}$.

We define the radix–2^q unshuffle permutation, $\Pi_{unshf,2^q}^{m,l}(a)$, in terms of the shuffle permutation as follows: $\Pi_{unshf,2^q}^{m,l}(a) = \Pi_{shf,2^{n-q}}^{m,l}(a)$, where $n = m - l + 1$.

The butterfly permutation, $\Pi_{bfly,Q}$, where $\log Q \in I$, the identity mapping, is a permutation on A_r such that $\Pi_{bfly,Q}(\mathbf{a}) = a \oplus Q$, where $\oplus$ is the addition on $\mathbb{F}_2$.

Which FFT? The complex–to–complex *forward* DFT, expressed as a mapping between two N–point complex sequences, $\mathbf{h} = (h_0, \ldots, h_{N-1}) \in \mathbb{C}^N$ and $\tilde{\mathbf{h}} = (\tilde{h}_0, \ldots, \tilde{h}_{N-1}) \in \mathbb{C}^N$ is given by: $\mathcal{F}(h_k) = \tilde{h}_k = \sum_{j=0}^{N-1} h_j \, e^{\frac{2\pi i}{N} jk}$, $k = 0, 1, \ldots, N-1$, where i is $\sqrt{-1}$, $\omega_N = e^{\frac{2\pi i}{N}}$ is the principle N^{th} root of unity, and ω_N^{jk} are the corresponding twiddle factors. However, the term *parallel FFT* is overloaded in its many variations, each of which may affect the form of the output, and the computational, storage or communication complexities, while still conforming to the DFT definition [5, 6, 15]. For our purposes, it is sufficient to demonstrate our methodology for any such variation. Thus, arbitrarily and for the sake of clarity, we select the forward, unscaled, ordered, radix–2, one–dimensional FFT with precomputed twiddle factors. Likewise, we arbitrarily choose the decimation in time (DIT) bit–reversed FFT, for which we assume a block data allocation scheme, already selected by the defined vector space computational framework, summarized above.

Specification: The theoretical framework of [12] enables an algorithmic specification for orthogonal transforms that facilitates algorithmic representation, derivation and analysis. We refer to the integral part of the FFT computation, the DIT butterfly operation on all data points from 0 to $N-1$, with radix–Q, as $O_{t\_bfly,Q}^{N-1,0}$, an instance of which is portrayed in Figure 1(b) for two data points. We define $\omega_N^{twi(a)}$ to be the twiddle factor for the pair of elements x_a and x_{a+Q} in $O_{t\_bfly,Q}^{N-1,0}$, where $twi(a)$ denotes their twiddle index. Each instance of such a butterfly operation consists of one complex multiplication, addi-

tion and subtraction, as well as butterfly permutation, $\Pi_{bfly,Q}$. Performing m consecutive operations at stage s of the algorithm is expressed in the form of an ordered sequence of m such operations, $\left(O_{f_1(s)} \bullet O_{f_2(s)} \bullet \ldots \bullet O_{f_m(s)}\right) = \left(O_{f_i(s)}\right)_{i=1}^{m}$, where $f_i(s)$ is some discrete function of s. Hence, a stage–operation pair precisely describes the operations performed at stage s in the form $\left(s, \left(O_{f_i(s)}\right)_{i=1}^{m}\right)$. A sequence (with enforced ordering) of n similar stage–operation pairs will be denoted by $\left(g(j), \left(O_{f_i(g(j))}\right)_i\right)_{j=1}^{n}$, where $g(j)$ is some discrete function on j. Finally, the entire algorithm is specified as an ordered sequence of such stage–operation pairs enclosed in curly braces.

Complexity Metrics: We group and denote the complexity metrics as $\mathcal{A}$, $\mathcal{M}$, $\mathcal{C}$ and $\mathcal{I}$ for *arithmetic*, *memory*, *communication* and *load imbalance* (for any metric relative imbalance) complexities. We isolate various *types* within each class, say $\mathcal{A}_{type}$, and further refine the metrics to $\mathcal{A}_{type\_nmax}$ and $\mathcal{A}_{type\_navg}$, which stand for the *maximum*, and *average* number of units of $\mathcal{A}_{type}$ per *node*, whereas $\mathcal{A}_{type\_all}$ is for *all nodes*. In class $\mathcal{A}$, we compute the *real* arithmetic operations, and assume complex adds take 2 real adds, and a complex multiply comprises 3 real adds and 3 real multiplys. $\mathcal{A}_{ssteps}$ is the number of *supersteps* (sequence of computation steps not interruptible by communication) for the entire algorithm, and $\mathcal{A}_{steps}$ gives the total number of computational steps. $\mathcal{M}_{active\_*}$ and $\mathcal{M}_{precomp\_*}$ denote the number of real *active data* (read–write) and *precomputed data* (read-only) in memory, respectively. $\mathcal{C}_{steps}$ are the *total* number of *communication steps* for the entire algorithm; while $\mathcal{C}_{ssteps}$ gives the *total* number of *communication supersteps* (sequence of communication steps between computation supersteps) for the entire algorithm. $\mathcal{C}_{trans\_*}$ is the number of *real* element transfers performed across all communication steps, but is also decompose into real element transfers for the particular communication patterns, such as: $\mathcal{C}_{bfly\_*}$, $\mathcal{C}_{bitrev\_*}$, $\mathcal{C}_{xch\_*}$ and $\mathcal{C}_{shf\_*}$. The imbalance ratio $\mathcal{I}$ ranges from 0 (for perfect load–balance) and is less than 1. Assuming that real multiplys costs μ times real adds,
$\mathcal{I}_{\mathcal{A}} = \left(\left(\mathcal{A}_{add\_nmax} - \mathcal{A}_{add\_navg}\right) + \mu\left(\mathcal{A}_{mul\_nmax} - \mathcal{A}_{mul\_navg}\right)\right) / \left(\mathcal{A}_{add\_all} + \mu\mathcal{A}_{mul\_all}\right)$,
$\mathcal{I}_{\mathcal{M}\,type} = \left(\mathcal{M}_{type\_nmax} - \mathcal{M}_{type\_navg}\right) / \mathcal{M}_{type\_all}$, for $type \in \{active, precomp\}$,
$\mathcal{I}_{\mathcal{C}} = \left(\mathcal{C}_{trans\_nmax} - \mathcal{C}_{trans\_navg}\right) / \mathcal{C}_{trans\_all}$.

3 Direct Parallelization

This refers to the straight forward approach of evenly distributing the input data among the processor nodes, which is specified as: $\left\{\left(0,\ \Pi_{bitrev}^{r-1,0} \bullet O_{t\_bfly,2^0}^{N-1,0}\right),\ \left(s,\ O_{t\_bfly,2^s}^{N-1,0}\right)_{s=1}^{r-1}\right\}$. Figure 2 depicts the direct (DR) parallelization of a DIT, bit–reversed FFT of size 8 among 4 processors. Let the a^{th} data element, $0 \le a \le N-1$ reside in physical location z at stage

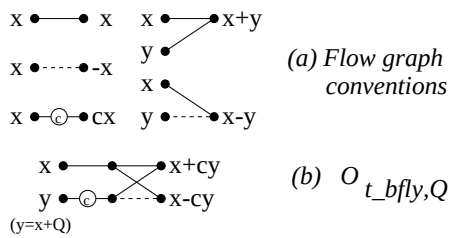

FIG. 1.

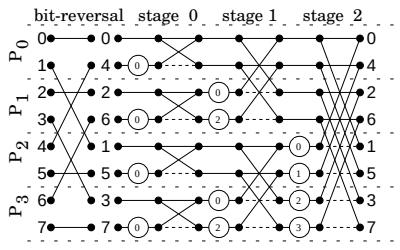

FIG. 2. *Direct Parallelization:* $N = 8$, $P = 4$

s. Then, the twiddle index, $twi(z) = twi\left(\left(\Pi_{bitrev}^{r-1,0}\right)^{-1}(a)\right) = (a_{r-1-s}) \cdot (a_{r-s} \ldots a_{r-2}a_{r-1}) \cdot$

2^{r-1-s}, which is for our bit–reversed DIT FFT. The first v stages are entirely local [12] (comprising a single computational superstep), whereas all subsequent p stages invoke communication (each constituting a computational superstep). The complexity metrics for direct parallelization are tabulated in Figures 4, 5, 6 and 7, which follow from the properties of $\Pi^{r-1,0}_{bitrev}$ and $\Pi_{bfly,Q}$ proved in [12]. These metrics illustrate that, for DR parallelization, while the communication is almost perfectly load–balanced among all the processor nodes (there is, in fact, a slight imbalance for $v < \lfloor r/2 \rfloor$), it is not the case with arithmetic and precomputed memory. Figure 2 demonstrates the source of this imbalance to stem from the multiplications at every stage, precisely what the load–balanced approach needs to remedy.

4 Load–balanced Parallelization

The goal is to achieve $\mathcal{I}_{\mathcal{A}} = \mathcal{I}_{\mathcal{M}_{active}} = \mathcal{I}_{\mathcal{M}_{precomp}} = \mathcal{I}_{\mathcal{C}} = 0$. In order to load balance the computation (and hence the precomputed storage requirements), we compel the butterfly operations to be local to the processor nodes at every stage of the algorithm [5, 15]. This is attained by placing the index bit currently operated on by the butterfly operation into the memory part of the index [12], thus guaranteeing $V/2$ multiplys per node. Such a *load–balancing permutation* (LBP) enforces this locality for one or more stages, necessitating another LBP whenever the locality "runs out". As a result, the output is scrambled, and an *order restoring permutation* (ORP) is required at the end. Figure 3 illustrates

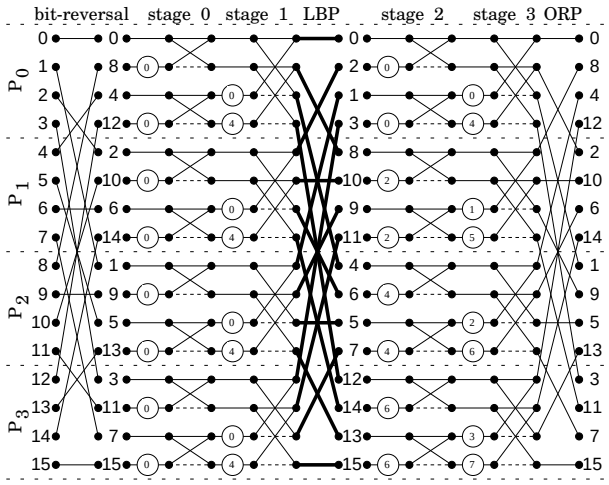

FIG. 3. *Radix–4 Load–balanced Parallelization: Case 1 ($V \geq P$), $N = 16$, $P = 4$*

the load–balanced (LB) approach for $V = 4$, where it is evident that the LBP is, in fact, a radix–4 exchange permutation [12]. However, a sequence of two radix–2 exchange permutations would produce exactly the same outcome. A result in [12] proves that the optimal solution is to perform one exchange permutation of the maximum possible radix rather than a sequence with lower radices. Clearly, the maximum attainable radix for a LBP is $\min\{P, V\}$.

In deriving the algorithm, the degrees of freedom (more than one permutation produces the desired result) are resolved according to the following criteria: (1) Memory bits not exchanged do not change their position within the index, thereby avoiding memory moves for elements remaining local to the processor after the LBP. (2) The block of bits to be exchanged moves into the least significant positions of the memory index, thereby minimizing the strides of the sequence of butterfly operations following the LBP. This

maximizes the probability for both operands of a butterfly operation to fall within the same cache fetch, which would be the case if the butterfly stride is less than half the cache size [3]. (3) An algorithmic pattern with a monotonically changing step repeats for as long as possible, which reduces index recalculation (indices usually serve as variables in a program loop), facilitates programming, and improves performance in most cases, especially if pipelining is in place [5].

For a rigorous derivation of the LB algorithm, refer to [10], which relies on the permutation results proved in [12]. Here, our exposition is limited to the derived specification. Since the twiddle index $twi(z)$ for any location z holding a data element is the same as for DR parallelization for stages $0 \le s \le v-1$, we only specify it for stages $v \le s \le r-1$. The specification of the LB algorithm is broken up into three cases:

Case 1 ($\boldsymbol{V \ge P}$):

$$\left\{ \left(0,\ \Pi^{r-1,0}_{bitrev} \bullet O^{N-1,0}_{t\_bfly,2^0}\right),\ \left(s,\ O^{N-1,0}_{t\_bfly,2^s}\right)_{s=1}^{v-1},\right.$$
$$\left. \left(v,\ \Pi^{v,0}_{xch,2^p} \bullet O^{N-1,0}_{t\_bfly,2^0}\right),\ \left(v+j,\ O^{N-1,0}_{t\_bfly,2^j}\right)_{j=1}^{p-1},\ \left(r,\ \Pi^{v,0}_{xch,2^p}\right)\right\}.$$

For stages $v+j$, $0 \le j \le p-1$, $twi(z) = twi\left((\Pi^{r-1,0}_{bitrev} \odot \Pi^{v,0}_{xch,2^p})^{-1}(a)\right) =$
$(a_j) \cdot (a_{j-1} \ldots a_0\ a_{v-1} \ldots a_p\ a_{r-1} \ldots a_v) \cdot 2^{p-1-j}$, where the bits $a_{j-1} \ldots a_0 = 1$ for $j = 0$.

Case 2 ($\boldsymbol{V < P}$, $\boldsymbol{p}$ mod $\boldsymbol{v} = \mathbf{0}$):

$$\left\{ \left(0,\ \Pi^{r-1,0}_{bitrev} \bullet O^{N-1,0}_{t\_bfly,2^0}\right),\ \left(s,\ O^{N-1,0}_{t\_bfly,2^s}\right)_{s=1}^{v-1},\right.$$
$$\left. \left[\left(kv,\ \Pi^{kv,\,0}_{xch,2^v} \bullet O^{N-1,0}_{t\_bfly,2^0}\right),\ \left(kv+j,\ O^{N-1,0}_{t\_bfly,2^j}\right)_{j=1}^{v-1}\right]_{k=1}^{p/v},\left(r,\ \Pi^{r-1,0}_{unshf,2^v}\right)\right\}.$$

For stages $kv+j$, $0 \le j \le p-1$, $1 \le k \le p/v$, $twi(z) =$
$twi\left(\left(\Pi^{r-1,0}_{bitrev} \odot (\odot_{k=1}^{p/v} \Pi^{kv,0}_{xch,2^v})\right)^{-1}(a)\right) = (a_j) \cdot (a_{j-1} \ldots a_0\ a_{(k+1)v-1} \ldots a_v) \cdot 2^{p-1-j}$, where the bits $a_{j-1} \ldots a_0 = 1$ for $j = 0$.

Case 3: ($\boldsymbol{V < P}$, $\boldsymbol{p}$ mod $\boldsymbol{v} \neq \mathbf{0}$):

$$\left\{\left(0, \Pi^{r-1,0}_{bitrev} \bullet O^{N-1,0}_{t\_bfly,2^0}\right), \left(s, O^{N-1,0}_{t\_bfly,2^s}\right)_{s=1}^{v-1}, \left[\left(kv, \Pi^{kv,\,0}_{xch,2^v} \bullet O^{N-1,0}_{t\_bfly,2^0}\right), \left(kv+j, O^{N-1,0}_{t\_bfly,2^j}\right)_{j=1}^{v-1}\right]_{k=1}^{\lfloor p/v \rfloor},\right.$$
$$\left(\left(\lfloor p/v \rfloor + 1\right)v,\ \Pi^{r-p \bmod v,\,0}_{xch,2^{p \bmod v}} \bullet O^{N-1,0}_{t\_bfly,2^0}\right),\ \left(\left(\lfloor p/v \rfloor + 1\right)v + j, O^{N-1,0}_{t\_bfly,2^j}\right)_{j=1}^{p \bmod v - 1},$$
$$\left.\left(r,\ \Pi^{v-1,0}_{unshf,2^{p \bmod v}} \odot \Pi^{r-1,0}_{unshf,2^v}\right)\right\}.$$

For stages $kv+j$, $0 \le j \le p-1$, $1 \le k \le \lfloor p/v \rfloor$, $twi(z) =$
$twi\left(\left(\Pi^{r-1,0}_{bitrev} \odot (\odot_{k=1}^{\lfloor p/v \rfloor} \Pi^{kv,0}_{xch,2^v})\right)^{-1}(a)\right) = (a_j) \cdot (a_{j-1} \ldots a_0\ a_{(k+1)v-1} \ldots a_v) \cdot 2^{p-1-j}$, where the bits $a_{j-1} \ldots a_0 = 1$ for $j = 0$.

For stages $(\lfloor p/v \rfloor + 1)v + j$, $0 \le j \le p \bmod v - 1$, $1 \le k \le \lfloor p/v \rfloor$,
$twi(z) = twi\left(\left(\Pi^{r-1,0}_{bitrev} \odot (\odot_{k=1}^{\lfloor p/v \rfloor} \Pi^{kv,0}_{xch,2^v}) \odot \Pi^{r-p \bmod v,0}_{xch,2^{p \bmod v}}\right)^{-1}(a)\right) =$
$(a_j) \cdot (a_{j-1} \ldots a_0\ a_{v-1} \ldots a_{p \bmod v}\ a_{r-1} \ldots a_{\lfloor p/v \rfloor + 1)v}\ a_{(\lfloor p/v \rfloor + 1)v-1} \ldots a_v) \cdot 2^{p-1-j}$, where the bits $a_{j-1} \ldots a_0 = 1$ for $j = 0$.

The complexity metrics of the LB approach are tabulated in Figures 4, 5, 6 and 7, where we achieve $\mathcal{I}_{\mathcal{A}} = \mathcal{I}_{\mathcal{M}_{active}} = \mathcal{I}_{\mathcal{M}_{precomp}} = \mathcal{I}_{\mathcal{C}} = 0$ (except for the case $V < P$, where a slight imbalance occurs in the bit-reversal, for both algorithms). For the performance metrics of case 3 ($V < P$, $p \bmod v \neq 0$), the subcase $v \le \lfloor \frac{r}{2} \rfloor$, as well as for a detailed derivation of the entire set of performance metrics, refer to [10].

Arithmetic	Direct	Load–bal
$\mathcal{A}_{steps}$	$2\log N$	$2\log N$
$\mathcal{A}_{super\_steps}$	$\log P + 1$	$\lceil \frac{\log N}{\log V} \rceil$
$\mathcal{A}_{add\_nmax}$	$\frac{1}{2}V(7\log N + 3\log P)$	$\frac{7}{2}V\ \log N$
$\mathcal{A}_{add\_navg}$	$\frac{7}{2}V\ \log N$	$\frac{7}{2}V\ \log N$
$\mathcal{A}_{mul\_nmax}$	$\frac{3}{2}V(\log N + \log P)$	$\frac{3}{2}V\ \log N$
$\mathcal{A}_{mul\_navg}$	$\frac{3}{2}V\ \log N$	$\frac{3}{2}V\ \log N$
$\mathcal{I}_{\mathcal{A}}$	$\frac{3(1+\mu)\ \log P}{(7+3\mu)\ P \log N}$	0

FIG. 4. *Parallel FFT Arithmetic Complexity*

Memory	Direct	Load–bal
$\mathcal{M}_{active\_nmax}$	$2V$	$2V$
$\mathcal{M}_{active\_navg}$	$2V$	$2V$
$\mathcal{I}_{\mathcal{M}_{active}}$	0	0
$\mathcal{M}_{precomp\_nmax}$	$V(\log N + \log P)$	$V\ \log N$
$\mathcal{M}_{precomp\_navg}$	$V\ \log N$	$V\ \log N$
$\mathcal{I}_{\mathcal{M}_{precomp}}$	$\frac{\log P}{P\ \log N}$	0

FIG. 5. *Parallel FFT Memory Complexity*

Commun.	Direct	Load–balanced
$\mathcal{C}_{ssteps}$	$\log P + 1$	3
$\mathcal{C}_{bfly\_nmax}$	$V \log P$	0
$\mathcal{C}_{bfly\_navg}$	$V \log P$	0
$\mathcal{C}_{bitrev\_nmax}$	$V(1-1/P)$	$V(1-1/P)$
$\mathcal{C}_{bitrev\_navg}$	$V(1-1/P)$	$V(1-1/P)$
$\mathcal{C}_{xch\_nmax}$	0	$2V(1-1/P)$
$\mathcal{C}_{xch\_navg}$	0	$2V(1-1/P)$
$\mathcal{C}_{shf\_nmax}$	0	0
$\mathcal{C}_{shf\_navg}$	0	0
$\mathcal{C}_{trans\_nmax}$	$V(p+1-1/P)$	$3V(1-1/P)$
$\mathcal{C}_{trans\_navg}$	$V(p+1-1/P)$	$3V(1-1/P$
$\mathcal{I}_{\mathcal{C}}$	0	0

FIG. 6. *Parallel FFT Communication Complexity for* $V \geq P$

Commun.	Direct	Load–balanced
$\mathcal{C}_{ssteps}$	$\log P + 1$	$r/v + 1$
$\mathcal{C}_{bfly\_nmax}$	$V \log P$	0
$\mathcal{C}_{bfly\_navg}$	$V \log P$	0
$\mathcal{C}_{bitrev\_nmax}$	V	V
$\mathcal{C}_{bitrev\_navg}$	$V - 2^{\lceil \frac{r}{2} \rceil}/P$	$V - 2^{\lceil \frac{r}{2} \rceil}/P$
$\mathcal{C}_{xch\_nmax}$	0	$\frac{p}{v}(V-1)$
$\mathcal{C}_{xch\_navg}$	0	$\frac{p}{v}(V-1)$
$\mathcal{C}_{shf\_nmax}$	0	V
$\mathcal{C}_{shf\_navg}$	0	$N - V$
$\mathcal{C}_{trans\_nmax}$	$V(p+1)$	$(2+\frac{p}{v})V - \frac{p}{v}$
$\mathcal{C}_{trans\_navg}$	$V(p+1)$ $-\frac{1}{P}2^{\lceil \frac{r}{2} \rceil} - \frac{p}{v}$	$(2+\frac{p}{v})V$ $-\frac{1}{p}(2^{\lceil \frac{r}{2} \rceil}+1)$
$\mathcal{I}_{\mathcal{C}}$	$\frac{2^{\lceil \frac{r}{2} \rceil}}{P(N(1+p)-2^{\lceil \frac{r}{2} \rceil})}$	$\frac{\frac{1}{p}(2^{\lceil \frac{r}{2} \rceil}+1)}{(2+\frac{p}{v})N - 2^{\lceil \frac{r}{2} \rceil} - V - \frac{p}{v}P}$

FIG. 7. *Parallel FFT Communication Complexity for* $V < P$ *and* $v \leq \lfloor \frac{r}{2} \rfloor$

5 Performance Analysis within Different Computational Models

We claim that our methodology for algorithmic derivation and specification adroitly delivers a refined set of complexity metrics applicable to a wide range of computational models and implementation environments. For instance, if all communication operations incur the same cost, regardless of their pattern, then the goal becomes minimizing $\mathcal{C}_{ssteps}$. Consulting Figures 6 and 7 shows that the LB approach wins for systems where $P > 4$ for the case $V \geq P$, and whenever $V \geq 4$ for the case $V < P$. On the other hand, if the communication cost is incurred by the total number of transfers between nodes, then our criteria becomes $\mathcal{C}_{trans\_all}$, revealing that, for the case $V \geq P$, the LB approach is superior whenever $P \geq 4$.

To illustrate how these metrics translate into some well known formal models, let the time taken by an algorithm be $T_{\mathcal{M}}$ time units, for a machine/model $\mathcal{M}$. First, consider the PRAM model [4]. Since all memory accesses are considered uniform, $T_{\mathcal{PRAM}} = \mathcal{A}_{add\_nmax} + \mu\mathcal{A}_{mul\_nmax}$, yielding $(7 + 3\mu)\frac{V}{2}\log N + (1 + \mu)\frac{V}{2}\log P$ and $(7 + 3\mu)\frac{V}{2}\log N$ for the DR and LB methods, respectively, thus favoring the latter for its arithmetic performance. An alternative model, the LPRAM [1] introduces communication via the machine latency parameter l for accessing global memory. Considering the more general case of $V \geq P$ for conciseness, $T_{\mathcal{LPRAM}} = T_{\mathcal{PRAM}} + l \cdot \mathcal{C}_{trans\_nmax}$, yielding $(7+3\mu)\frac{V}{2}\log N + (1+\mu)\frac{V}{2}\log P + l \cdot 3V(\log P + 1 - 1/P)$ and $(7+3\mu)\frac{V}{2}\log N + l \cdot 3V(1-1/P)$ for the DR and LB methods, respectively, confirming that the LB method wins.

Within the realm of the BSP [17], we let the four model parameters be N and P as before, g the ratio between communication and computation operations, and L the synchronization cost. In such a scenario, for the case $V \geq P$, and following the guidelines in [14], $T_{\mathcal{BSP}} = \mathcal{A}_{add\_nmax} + \mu\mathcal{A}_{mul\_nmax} + g \cdot \mathcal{C}_{bitrev\_nmax} + L + g \cdot \sum_{s=v}^{r-1} \mathcal{C}_{bfly\_nmax}(s) +$

$\mathcal{C}_{ssteps} \cdot L$ for the DR method, and $T_{\mathcal{BSP}} = \mathcal{A}_{add\_nmax} + \mu\mathcal{A}_{mul\_nmax} + g \cdot \mathcal{C}_{bitrev\_nmax} + g \cdot \sum_{s=1}^{2} \mathcal{C}_{xch\_nmax}(s) + \mathcal{C}_{ssteps} \cdot L$ for the LB method. A simple analysis suggests that in terms of communication cost, the LB outperforms the DR whenever $V \geq P/2$, which is always true for $V \geq P$. The more refined LogP model [3] assumes message passing as the general paradigm, providing flexibility of performance analysis for very specialized implementations. In our case, assuming the natural barrier synchronization after every superstep, we may bundle the overhead per message parameter o with the gap g. Thus, for our purposes, $T_{\mathcal{LOGP}} = T_{\mathcal{BSP}|g \leftarrow g+o}$, resulting in the same conclusion as the BSP analysis.

6 A Comparative Implementation

A data parallel CM–Fortran implementation was carried out on the CM–5 (where P is the number of vector units), for the purpose of comparing the two methods. The intent was neither to compete with highly tuned library codes, (implemented at a lower level and performing higher radix–4 FFTs locally to boost performance), nor to contrast this against other paradigms, such as message passing (with access to finer grain machinery). Rather, emphasis was made on a fair comparison with uniform assumptions and optimizations for each case. Purely radix–2 FFTs were computed, with a customization of the first two stages to minimize arithmetic. For B disjoint butterfly blocks, we control the geometry by aliasing the N–size 1D data array $\mathbf{h}(N = P \times V)$, as a 2D array $\mathbf{h}(P \times \frac{B}{P}, 1 \times \frac{N}{B})$ for $B \geq P$, and as $\mathbf{h}(B \times 1, \frac{B}{P} \times V)$ for $B < P$, to perform efficient butterfly operations that only invoke one *cshift*. This gives a tremendous advantage to the DR version, setting the cost of $\Pi^{m,l}_{xch,S}$ and $\Pi^{m,l}_{shf,S}$, which are performed via a global send through the router, at five to ten times that of $\Pi_{bfly,Q}$, depending on Q. Moreover, unlike *cshift*, all permutations implemented via the general send, including $\Pi^{m,l}_{bitrev}$, require an address calculation phase.

To analyze the relative performance, the code is highly instrumented, which not only introduces substantial time dilation, but also breaks up the pipelining capability of the compiler, preventing the code from achieving near peak FLOP rate performance.

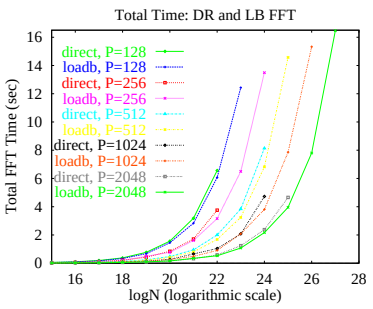

FIG. 8. *Total Times*

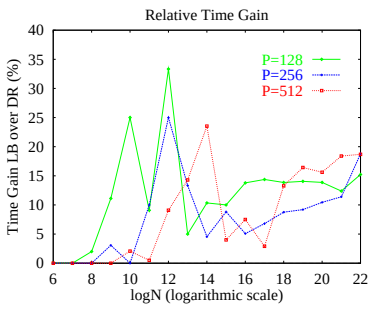

FIG. 9. *Relative Gain*

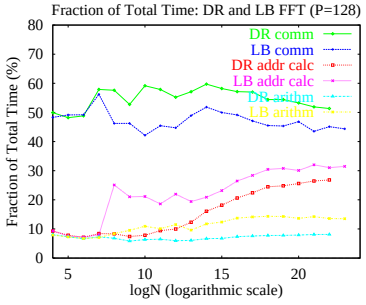

FIG. 10. *Fractions of Time*

As prophecied by $\mathcal{M}_{precomp\_nmax}$, Figure 8 illustrates that the LB version computes problem sizes at least twice or four times as large as the DR version, for the entire set of processor values. Moreover, despite the advantages of the DR implementation discussed above, the LB algorithm consistently outperforms its direct counterpart.

Figure 9 is a partial magnification of Figure 8, showing the relative time gain of the LB over the DR algorithm for $P = 128, 256,$ and 512. If t_{DR} and t_{LB} are the total times of the DR and LB approaches, respectively, the relative ratio $(t_{DR} - t_{LB})/t_{LB}$ is depicted. Note that the LB version is identical to the DR for $V = 1$, when the relative gain is zero. Then,

we observe a transitional period, where the relative gain oscillates from approximately 5% to 35%, which results from the uneven performance of the LB method for cases 2, $V < P$, $p \bmod v = 0$ and 3, $V < P$, $p \bmod v \neq 0$. This transitional period lasts from $\log N$ sizes of 8 to 14, 9 to 16, and 10 to 18 for the curves $P = 128, 256$, and 512, respectively. After that, case 1, $V \geq P$, is entered, where the relative performance gain settles around 15–20%.

In Figure 10, we plot three different curves per algorthm, which account for the fraction of the total time spent on communication, address calculation and arithmetic, for $P = 128$. The implementational disadvantage of the LB algorithm versus the DR, only calculating addresses for $\Pi^{r-1,0}_{bitrev}$, is illustrated by the higher fraction of time that the LB spends in address calculation, the difference being 10% on the average. Despite this disadvantage, the LB algorithm still spends less of its time communicating, about 50% as opposed to 60% for the DR. Moreover, the LB approach spends a higher fraction of its time in arithmetic computation, with a 10% lead over the DR, proving that the LB algorithm is more efficient.

We have illustrated that the LB approach is a significant win for an FFT kernel, from all perspectives: it is faster, requires less memory, and despite the advantage of the DR approach within this implementation, spends a smaller fraction of its time communicating. These observations agree closely with the analytical performance metrics.

References

[1] A. Aggarwal, A. K. Chandra, and M. Snir, *Communication complexity of PRAMs*, Theoretical Computer Science, 71 (1990), pp. 3–28.

[2] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Mathematics of Computation, 19 (1965), pp. 297–301.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: Towards a realistic model of parallel computation*, in 1993 Procedings of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 1993.

[4] S. Fortune and J. Wyllie, *Parallelism in random access machines*, in Proceedings of the 10th Annual Symposiym on Theory of Computing, 1978, pp. 114–118.

[5] S. L. Johnsson, M. Jacquemin, and R. L. Krawitz, *Communication efficient multi-processor FFT*, Journal of Computational Physics, 102 (1992), pp. 381–397.

[6] S. L. Johnsson and R. L. Krawitz, *Cooley-Tukey FFT on the Connection Machine*, Parallel Computing, 18 (1992), pp. 1201–1221.

[7] S. L. Johnsson, R. L. Krawitz, D. MacDonald, and R. Frye, *A radix-2 FFT on the Connection Machine*, in Supercomputing 89, ACM, November 1989, pp. 809–819.

[8] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, 1982.

[9] R. B. Pelz, *Parallel compact FFTs for real sequences*, SIAM Journal of Scientific Computing, 14 (1993), pp. 914–935.

[10] N. Shalaby, *Efficient parallel FFTs: A generalized approach*, Tech. Rep. TR–33–95, Harvard University, Nov. 1995.

[11] ——, *Parallel discrete cosine transforms: Theory and practice*, Tech. Rep. TR–34–95, Harvard University, Dec. 1995.

[12] N. Shalaby and S. L. Johnsson, *A vector space framework for parallel stable permutations*, Tech. Rep. TR–32–95, Harvard University, Nov. 1995. (submitted for publication).

[13] ——, *Hierarchical load balancing for parallel fast Legendre transforms*, in Eighth SIAM Conference for Parallel Processing for Scientific Computing, 1997. (to appear).

[14] D. Skillicorn, J. M. Hill, and W. McColl, *Questions and answers about bsp*, Tech. Rep. PRG–TR–15-96, Oxford Unviverstiy Computing Laboratory, Parks Road, Oxford OX1 3QD, 1996.

[15] P. Swarztrauber, *Multiprocessor FFTs*, Parallel Computing, (1987), pp. 197–210.

[16] C. Temperton, *Self sorting in-place fast Fourier Transforms*, SIAM Journal of Sci. Stat. Computation, 12 (1991), pp. 808–823.

[17] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.